數位影像處理 作業一 報告
網媒所碩一 R11944045 呂佳晏

Bilinear & Bicubic interpolation:

• which function you use or implement

```
import cv2

img = cv2.imread("origin.jpg")
img1 = cv2.resize(img, None ,fx = 0.2, fy = 0.2, interpolation = cv2.INTER_LINEAR)
img2 = cv2.resize(img, None ,fx = 5, fy = 5, interpolation = cv2.INTER_LINEAR)
img3 = cv2.resize(img, None ,fx = 32, fy = 32, interpolation = cv2.INTER_LINEAR)
img4 = cv2.resize(img, None ,fx = 0.2, fy = 0.2, interpolation = cv2.INTER_CUBIC)
img5 = cv2.resize(img, None ,fx = 5, fy = 5, interpolation = cv2.INTER_CUBIC)
img6 = cv2.resize(img, None ,fx = 32, fy = 32, interpolation = cv2.INTER_CUBIC)


cv2.imwrite("bilinear_0.2.jpg", img1)
cv2.imwrite("bilinear_5.jpg", img2)
cv2.imwrite("bilinear_32.jpg", img3)
cv2.imwrite("bicubic_0.2.jpg", img4)
cv2.imwrite("bicubic_5.jpg", img5)
cv2.imwrite("bicubic_32.jpg", img6)
```

• how does my program work

我們可以利用 OpenCV 提供的 resize() 函數來改變圖片的大小，同時 resize() 函數

也可以讓我們選擇插值的方法，在這個作業中，我們選擇的是

cv2.INTER_LINEAR 以及 cv2.INTER_CUBIC 來達成作業的要求

函數的原型是：`cv2.resize(src, dsize[, dst[, fx[, fy[, interpolation]]]])`

fx, fy 分別代表可以放縮放的比例，在這份作業中

我們可以將 0.2、5、32 分別代入即可得出結果

cv2.INTER_LINEAR 代表利用 bilinear interpolation 來縮放圖片

cv2.INTER_CUBIC 代表利用 bicubic interpolation 來縮放圖片

• how to use my program

作業系統 : macOS Monterey 12.6

Python 版本：python 3.9.12

OpenCV 版本：4.6.0.66

在終端機輸入以下指令

pip3 install opencv-python

python3 hw1.py

結果會自動生成在程式所在的資料夾

Bilinear interpolation & Bicubic interpolation result comparison

• original image

• scaling factor 0.2

Bilinear

Bicubic

• scaling factor 5

Bilinear

Bicubic

• scaling factor 32

Bilinear

Bicubic

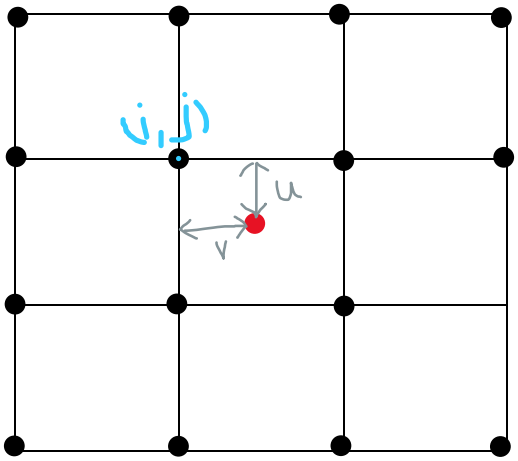
由上面三種不同的 scaling factor，經由 bilinear interpolation 跟 bicubic interpolation 操作後得出的結果圖片來看，bicubic interpolation 的結果相較於 bilinear interpolation 品質較高。

scaling factor 在 5 和 32 時，bicubic interpolation 的結果較為滑順，相比之下經由 bilinear interpolation 處理後的圖片 ，則顯略微模糊。

scaling factor 在 0.2 時，較沒有辦法清楚比較 bilinear interpolation 跟 bicubic interpolation 兩種插值方法的不同。

The method of Bicubic interpolation

bicubic interpolation 的做法是，利用目標點 N 周圍最近的 16 個點，透過計算這 16 的點的權重，再對這 16 個點計算加權平均數，求出點 N 的值。



假設目標點為(i + u, v + j)，那我們可以透過計算，求出離他最近的 16 個點，將這些點的權重加權起來可以用數學式表示為：

$$f(i+u,j+v)=\sum_{row=-1}^{2}\sum_{col=-1}^{2}f(i+row,j+col)W(row-u)W(col-v)$$

其中$f(i+u,j+v)$ 代表的是 (i + u, v + j) 這個點的值，$W(row-u)$ 和 $W(col-v)$ 則是插值的計算公式，根據維基百科，我們可以透過

$$W(x)=\begin{cases}(a+2)|x|^3-(a+3)|x|^2+1, & for\ |x|\le 1,\\ a|x|^3-5a|x|^2+8a|x|-4a, & for\ 1<|x|<2\\ 0, & otherwise\end{cases}$$

這個公式，來協助求出加權值，這個公式可以讓 W(0) = 1 以及 W(n) = 0。

The computational complexity of Bicubic and Bilinear interpolation

假設原本的圖片長為 M 寬為 N，scaling factor 為常數 s

・bilinear interpolation

處理後的圖片長 s * M 寬 s * N，需要對每一點進行三次運算，橫列縱列個插值一次，因此 computational complexity 為 O(MN)，總計算量約為 3 * M * N

・bicubic interpolation

處理後的圖片長 s * M 寬 s * N，需要對三個維度的每個點進行計算，而每個計算中，需要利用到矩陣乘法，因此總計算量約為 3 * M * N * (4 ^ 3)，時間複雜度雖然也是 O(MN)，但是實際需要的時間約為 bilinear interpolation 的 64 倍。

Note:

原本自己是手寫出這兩個方法的函式，但是發現 OpenCV 有提供內建的函式可以呼叫，因此作業的結果以及程式碼是用 OpenCV 提供的函式來進行

運算的分析則是以自己寫的函式來分析，故我在報告的最後兩頁附上自己寫的函式以供參考

Reference:

[1] https://en.wikipedia.org/wiki/Bicubic_interpolation

[2] https://www.twblogs.net/a/5ee5256006527f2d10e1b6e9

[3] https://dailc.github.io/2017/11/01/imageprocess_bicubicinterpolation.html

• Bilinear interpolation function code

```
#imple Bilinear Interpolation
def bilinear(image, scaling_factor):
    row, col, dim = image.shape
    new_row = int(row * scaling_factor)
    new_col = int(col * scaling_factor)
    # create new image
    result = np.zeros((new_row, new_col, dim))

    # mapping to origin image
    for i in range(new_row):
        for j in range(new_col):
            x = i / scaling_factor
            y = j / scaling_factor

            #計算要 Bilinear Interpolation 的四條線
            x_floor = math.floor(x)
            x_ceil = min(row - 1, math.ceil(x))
            y_floor = math.floor(y)
            y_ceil = min(col - 1, math.ceil(y))

            #利用上述四條線獲得四周的端點
            top_left = image[x_floor, y_floor, : ]
            top_right = image[x_floor, y_ceil, : ]
            down_left = image[x_ceil, y_floor, : ]
            down_right = image[x_ceil, y_ceil, : ]

            # due to x & y may not be an integer > we should cast it to an integer thus we use(int)x
            if (x_ceil == x_floor) and (y_ceil == y_floor):
                q = image[int(x), int(y), :]
            elif (x_ceil == x_floor):
                q1 = image[int(x), int(y_floor), :]
                q2 = image[int(x), int(y_ceil), :]
                q = q1 * (y_ceil - x) + q2 * (y - x_floor)
            elif (y_ceil == y_floor):
                q1 = image[int(x_floor), int(y), :]
                q2 = image[int(x_ceil), int(y), :]
                q = (q1 * (x_ceil - y)) + (q2 * (x - y_floor))
            else:
                q1 = top_left * (y_ceil - y) + top_right * (y - y_floor)
                q2 = down_left * (y_ceil - y) + down_right * (y - y_floor)
                q = q1 * (x_ceil - x) + q2 * (x - x_floor)

            result[i, j, :] = q # assign result to the result image
    return result.astype(np.uint8)
```

利用 cv2 內建函式讀取欲處理的圖片，再將圖片以及 scaling factor 傳入自建的函式 bilinear 中做處理，bilinear 函式利用了 NumPy 處理陣列的功能，先依照 scaling factor 創建一個只處理邊長的新影像結果，再將新的影像中每一個由上到下由左到右的 pixel 除以 scaling factor 對應回原本的圖片中，以下簡稱點 N，接著因為點 N 的 (x, y) 座標不一定皆為整數，因此我們藉由 floor() 以及 ceil() 函式計算出點 N 的四周端點，找到四個點後，我們再對這四個點以及點 N 做 bilinear interpolation，在計算過程中，除了考慮邊界的問題，也會同時針對如果點 N 映射回來是在四周端點連起來的線上，而非中心的情況。

- Bicubic interpolation function code

```
def w(x, a = -0.5):
    if abs(x) <= 1:
        return (a + 2) * (abs(x) ** 3) - (a + 3) * (abs(x) ** 2) + 1
    elif abs(x) > 1 and abs(x) < 2:
        return a * (abs(x) ** 3) - (5 * a) * (abs(x) ** 2) + 8 * a * abs(x) - 4 * a
    else:
        return 0


def pad_around(image):
    row, col, dim = image.shape
    pad_image = np.zeros((row + 4, col + 4, dim))
    pad_image[2:row + 2, 2:col + 2] = image #middle
    pad_image[2:row + 2, 0:2]=image[:, 0:1] #left
    pad_image[2:row + 2, col + 2:col + 4]=image[:, col - 1:col] #right
    pad_image[0:2, 2:col + 2]=image[0:1, :] #top
    pad_image[row + 2:row + 4, 2:col + 2]=image[row - 1: row, :] #down
    pad_image[0:2, 0:2]=img[0, 0] #top-left
    pad_image[0:2, col + 2:col + 4]=img[0,col - 1] #top-right
    pad_image[row + 2:row + 4, 0:2]=img[row - 1,0] #down-left
    pad_image[row + 2:row + 4, col + 2:col + 4]=img[row - 1,col - 1]#down-right
    return pad_image


def bicubic(image, scaling_factor):
    row, col, dim = image.shape
    #pad around incase next step out of boundry
    image = pad_arrond(image)
    new_row = int(row * scaling_factor)
    new_col = int(col * scaling_factor)
    # create new image
    result = np.zeros((new_row, new_col, dim))
    # mapping to origin image

    for k in range(dim):
        for i in range(new_row):
            for j in range(new_col):
                x = i / scaling_factor
                y = j / scaling_factor

                #計算要 bicubic Interpolation 的四條線
                x_floor = math.floor(x)
                x_ceil = min(row - 1, math.ceil(x))
                y_floor = math.floor(y)
                y_ceil = min(col - 1, math.ceil(y))
                u = x - x_floor
                v = y - y_floor

                # due to x may not be an integer > we should cast it to an intege
                A = np.matrix([w(1 + u), w(u), w(1 - u), w(2 - u)])
                C = np.matrix([[w(1 + v)], [w(v)], [w(1 - v)], [w(2 - v)]])
                B = np.matrix([
                    [image[int(x_floor) - 1, int(y_floor) - 1][k], image[int(x_floor) - 1, int(y_floor)][k],
                     image[int(x_floor) - 1, int(y_ceil)][k], image[int(x_floor) - 1, int(y_ceil) + 1][k]],
                    [image[int(x_floor), int(y_floor) - 1][k], image[int(x_floor), int(y_floor)][k],
                     image[int(x_floor), int(y_ceil)][k], image[int(x_floor), int(y_ceil) + 1][k]],
                    [image[int(x_ceil), int(y_floor) - 1][k], image[int(x_ceil), int(y_floor)][k],
                     image[int(x_ceil), int(y_ceil)][k], image[int(x_ceil), int(y_ceil) + 1][k]],
                    [image[int(x_ceil) + 1, int(y_floor) - 1][k], image[int(x_ceil) + 1, int(y_floor)][k],
                     image[int(x_ceil) + 1, int(y_ceil)][k], image[int(x_ceil) + 1, int(y_ceil) + 1][k]]])

                result[i, j][k] = np.dot(np.dot(A, B), C)
    return result.astype(np.uint8)
```

利用 cv2 內建函式讀取欲處理的圖片，再將圖片以及 scaling factor 傳入自建的函式 bicubic 中做處理， bicubic 函式除了會先依照 scaling factor 創建一個只處理邊長的新影像結果，也會對原本的影像做處裡。

因為 bicubic 在運算中，是參考周圍 16 個點的權重，因此我們將原本的影像經由自建的 pad_around 函式四周各自延伸兩個 pixel，pixel 的值設成跟四周圍的 pixel 相同。接下來再回到函式 bicubic 將新的影像中每一個由上到下由左到右的 pixel 除以 scaling factor 對應回原本的圖片中，以下簡稱點 N，接著因為點 N 的 (x, y) 座標不一定皆為整數，因此我們藉由 floor() 以及 ceil() 函式計算出點 N 的四周端點，找到四個點後，就可以藉由這四個點再延伸到四周的 16 個端點，並透過 NumPy 套件進行矩陣計算計算得出點 N 的值。